Algoritmi e Strutture Dati

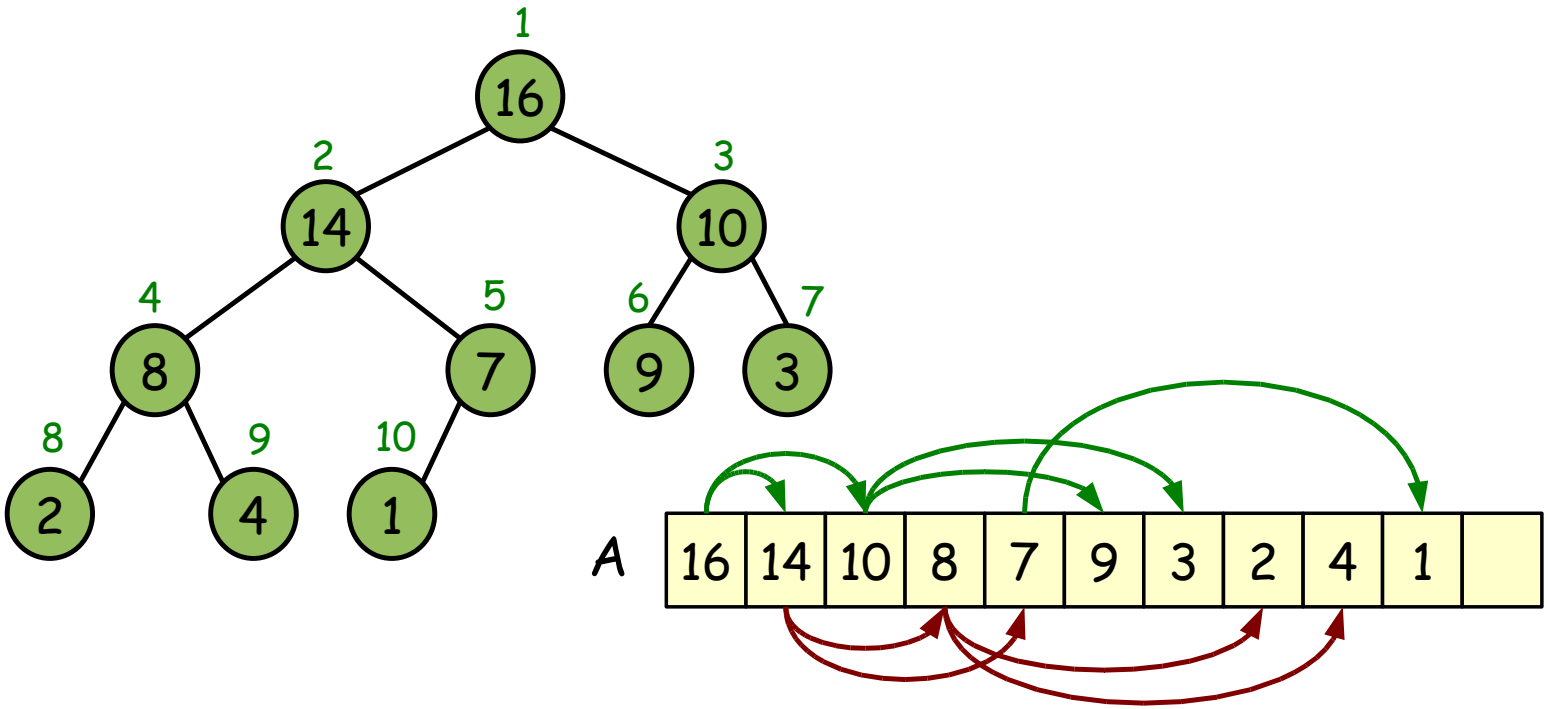
Heap

Maria Rita Di Berardini, Emanuela Merelli[1]

[1]Dipartimento di Matematica e Informatica
Università di Camerino

Heap

Un heap binario è una struttura dati composta da un array che può essere considerato come un albero binario quasi completo



Heap

Ogni nodo dell'albero corrisponde ad un elemento dell'array che memorizza il valore del nodo

Quasi completo significa che tutti i livelli, tranne l'ultimo, sono completi: possono mancare alcune foglie consecutive a partire dall'ultima foglia a destra

La "quasi" completezza garantisce che l'altezza di un heap con n nodi è $\Theta(\log_2 n)$

Vedremo che le operazioni fondamentali sugli heap vengono eseguite in un tempo che è al più proporzionale all'altezza dell'heap e, quindi, richiedono un tempo $O(\log_2 n)$

Heap

Un array che rappresenta un heap binario è un oggetto con due attributi:

- $length[A]$ è il numero di elementi (la dimensione) dell'array
- $heap\text{-}size[A]$ è il numero di elementi dell'heap memorizzati nell'array

La radice dell'albero è $A[1]$ (si trova in posizione 1)

Se i è l'indice di un dato nodo, gli indici del padre $parent[i]$, del figlio sinistro $left[i]$ e del figlio destro $right[i]$ sono $\lfloor i/2 \rfloor$, $2i$ e $2i + 1$

Max-heap e min-heap

Esistono due tipi di heap binari: max-heap e min-heap

In entrambi i tipi di heap binario, i valori dei nodi soddisfano una proprietà le cui caratteristiche dipendono dal tipo di heap

Proprietà del max-heap: in un **max-heap** ogni nodo i diverso dalla radice è tale che $A[parent[i]] \geq A[i]$

Nota 1: l'elemento più grande di un max-heap viene memorizzato nella radice

Nota 2: un sottoalbero di un nodo u contiene nodi il cui valore non è mai maggiore del valore del nodo u

Max-heap e min-heap

Proprietà del min-heap: in un **min-heap** ogni nodo i diverso dalla radice è tale che $A[parent[i]] \leq A[i]$

Nota 1: l'elemento più piccolo di un min-heap viene memorizzato nella radice

Nota 2: un sottoalbero di un nodo u contiene nodi il cui valore non è mai minore del valore del nodo u

Code di Priorità (Priority Queues)

Un **coda di priorità** è una struttura dati che serve per mantenere un'insieme S di elementi ciascun con un valore detto **chiave** che rappresenta la sua priorità

Analogamente agli heap, esistono due tipi di code di priorità: code di max-priorità e code di min-priorità

Una coda di max-priorità supporta le seguenti operazioni:

1. **Insert**(S, x) inserisce x nell'insieme S
2. **Maximum**(S) restituisce l'elemento di S con chiave più grande
3. **Extract-Max**(S) elimina e restituisce l'elemento di S con chiave più grande
4. **Increase-Key**(S, x, k) aumenta il valore della chiave di x al nuovo valore k (qui assumiamo $k \geq$ dell'attuale valore della chiave di x)

Code di Priorità (Priority Queues)

Applicazioni di una coda di max-priorità: programmare i lavori su un computer condiviso (Job-Scheduling)

La coda di max-priorità tiene traccia dei lavori da svolgere e delle relative priorità

Quando un lavoro è ultimato o interrotto viene selezionato, mediante **Extract-Max**, il lavoro con priotità più alta tra quelli in attesa

Un nuovo lavoro può essere aggiunto in qualsiasi momento mediante una **Insert**

Code di Priorità (Priority Queues)

In alternativa, una coda di min-priorità supporta le seguenti operazioni:

1. **Insert**(S, x) inserisce x nell'insieme S
2. **Minimum**(S) restituisce l'elemento di S con chiave minima
3. **Extract-Min**(S) elimina e restituisce l'elemento di S con chiave minima
4. **Decrease-Key**(S, x, k) decrementa il valore della chiave di x al nuovo valore k (qui assumiamo $k \leq$ dell'attuale valore della chiave di x)

Code di Priorità (Priority Queues)

Applicazioni di una coda di min-priorità: simulatore controllato di eventi (event-driven simualtion)

Gli elementi della coda sono gli eventi da simulare; ad ognuno di essi è associata una chiave che rappresenta il tempo in cui l'evento si verifica

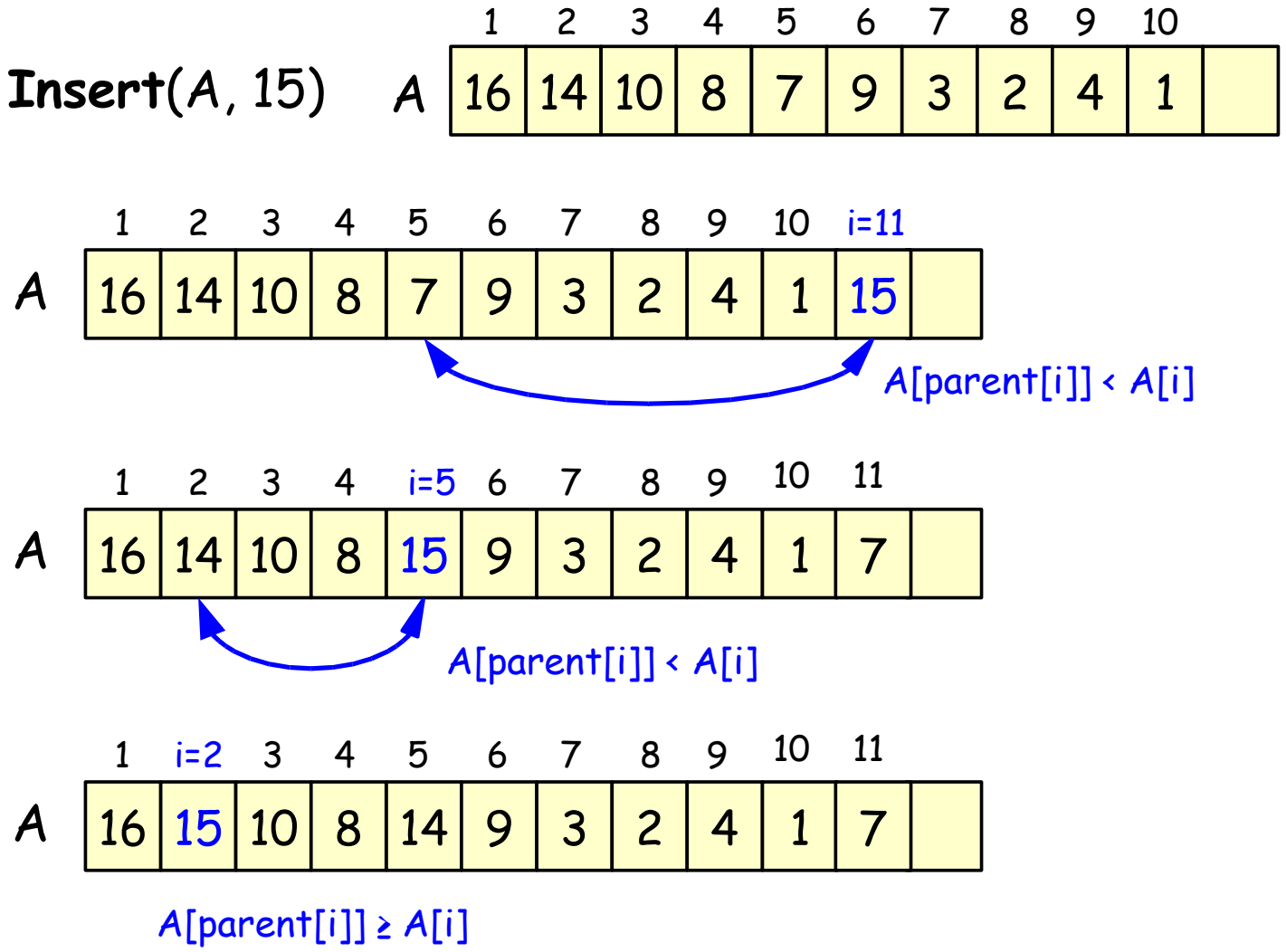
Gli eventi devono essere simulati secondo l'ordine dei loro tempi, questo perchè un evento può generare altri eventi

Ad ogni passaggio il programma seleziona l'evento da simulare mediante una **Etract-Min**

Ogni nuovo evento viene inserito in coda tramite una **Insert**

Vediamo come è possibile implementare una coda di max-priorità tramite un max-heap

Inserimento



Max-Heap-Insert(A, x)

Max-Heap-Insert(A, x)

$heap\text{-}size[\mathrm{A}] = \text{heap-size}[\mathrm{A}] + 1$

▷ aumentiamo la dimensione dell'heap

$i = heap\text{-}size[A]$

$A[i] \leftarrow x$

while $i > 1$ or $A[parent[i]] < A[i]$

do scambia $A[i] \leftrightarrow A[parent[i]]$

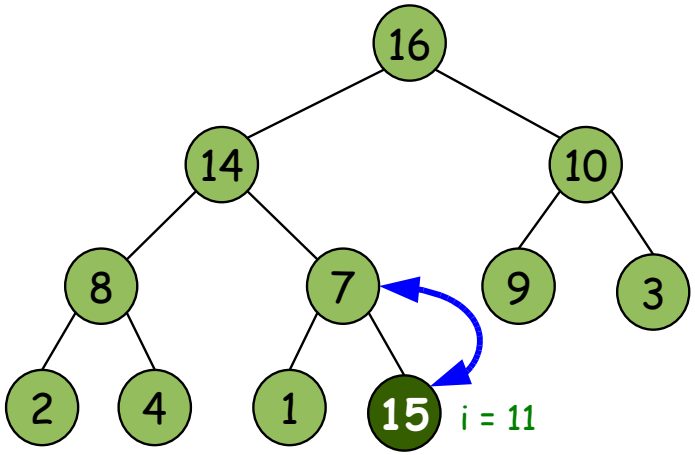
$i \leftarrow parent[i]$

Il numero di scambi è pari al più all'altezza dello heap, quindi

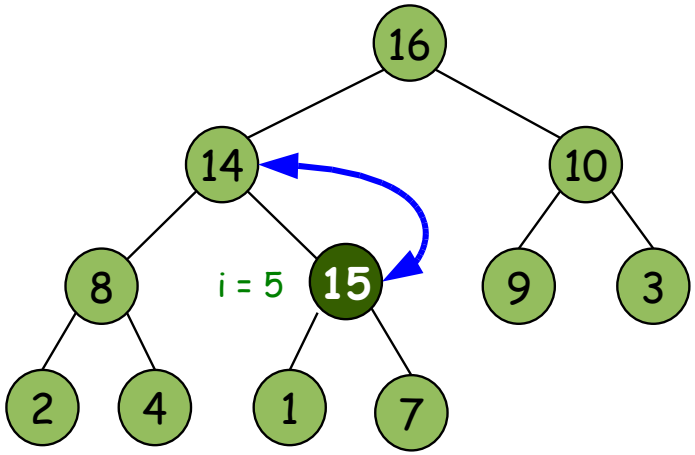
Max-Heap-Insert viene eseguita in un tempo

$$O(\log_2 n)$$

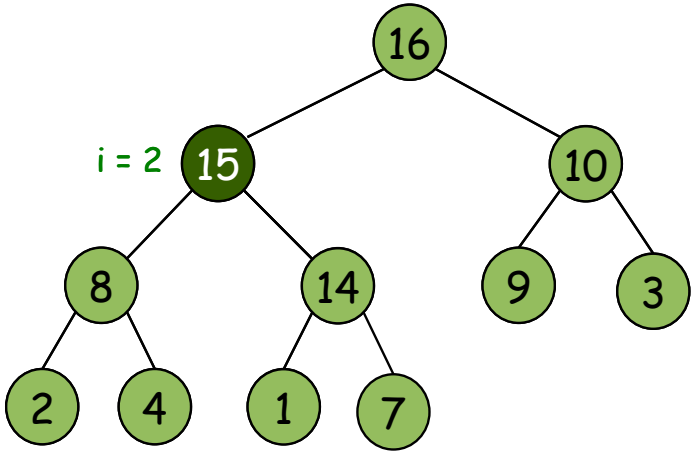
Max-Heap-Insert(A, x)



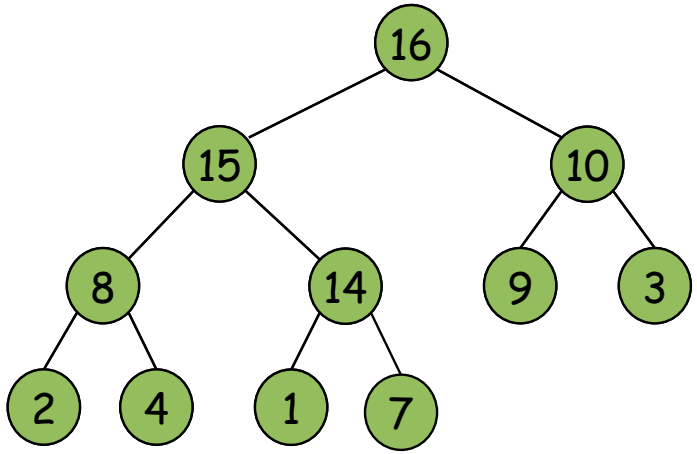
(a)



(b)



(c)

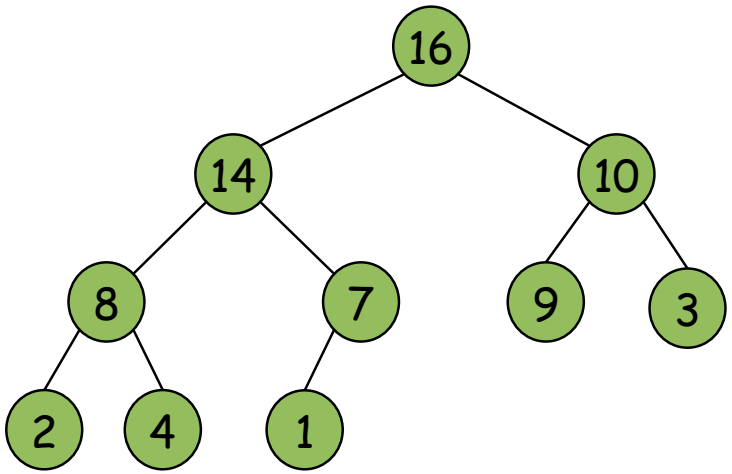


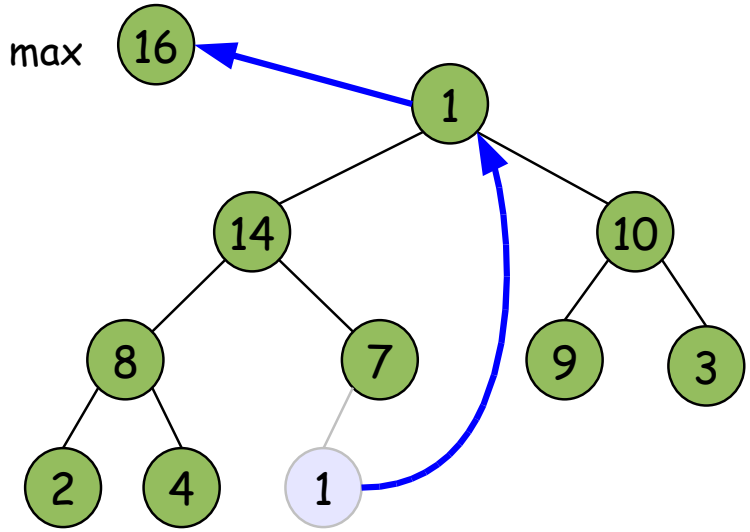
Restituzione del massimo

La seguente procedura implementa l'operazione **Maximum** in un tempo costante ($\Theta(1)$)

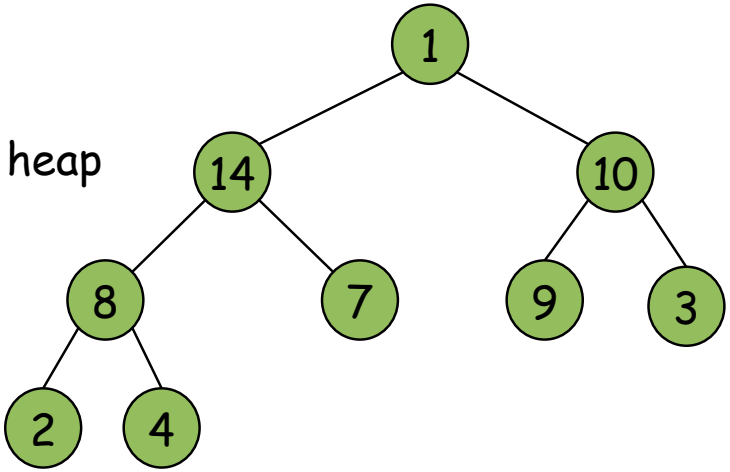
Heap-Maximum(A)
return $A[1]$

Estrazione del massimo





Non è un heap



Max-Heapify(*A*,1)

Max-Heapify(A, i)

È un importante subroutine per manipolare max-heap

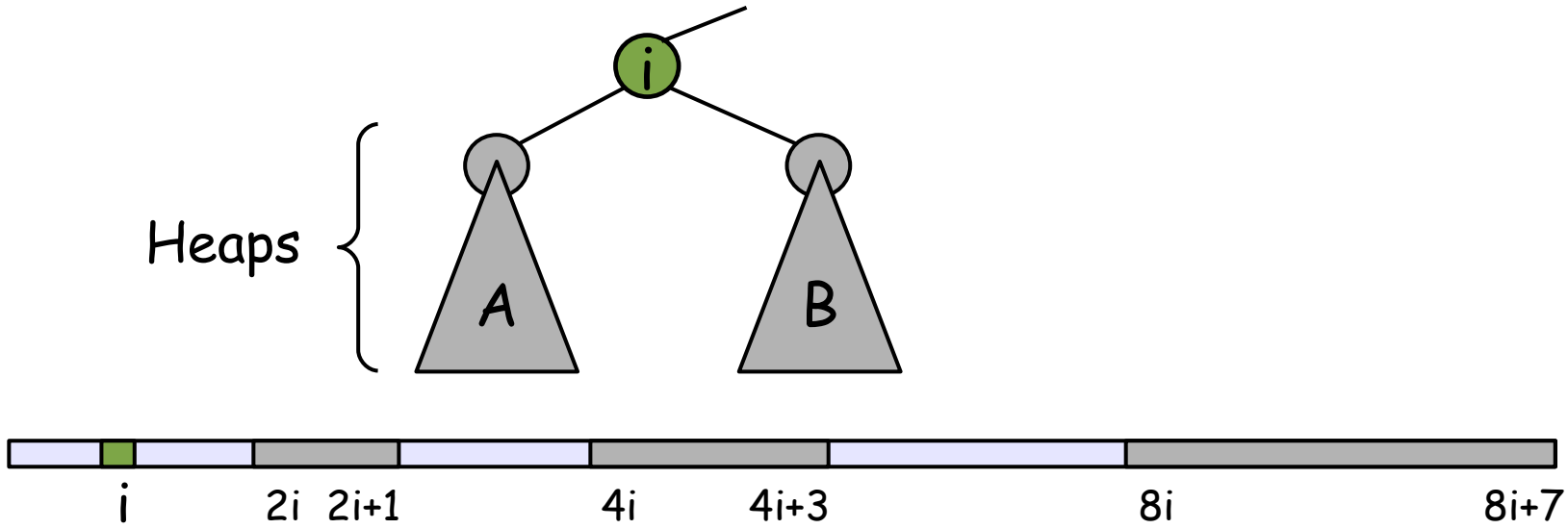
Input: un heap A e un indice i

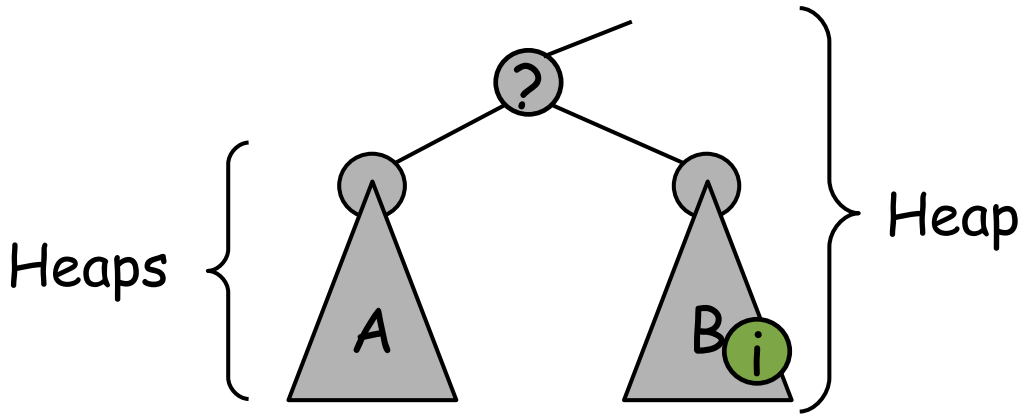
Max-Heapify(A, i) viene chiamata quando

- i sotto-alberi binari con radice $left[i]$ e $right[i]$ sono max-heap,
- ma $A[i]$ è più piccolo dei suoi figli, violando così la proprietà del max-heap

Idea: far scendere il valore di $A[i]$ nel max-heap in modo da ripristinare la proprietà desiderata

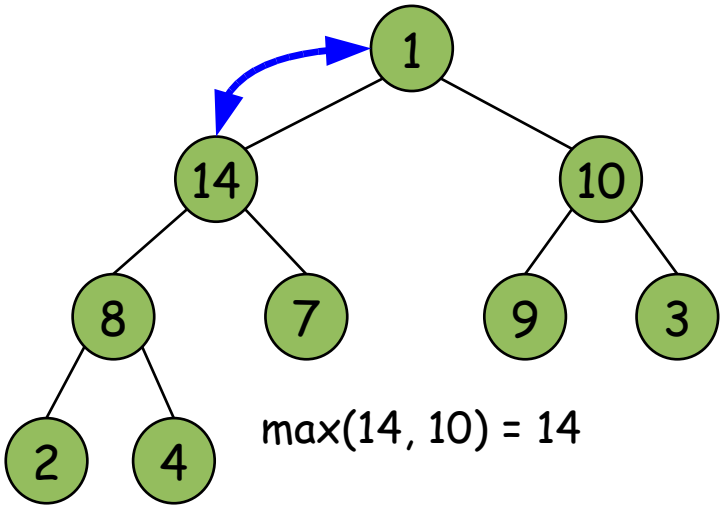
Max-Heapify(A, i)

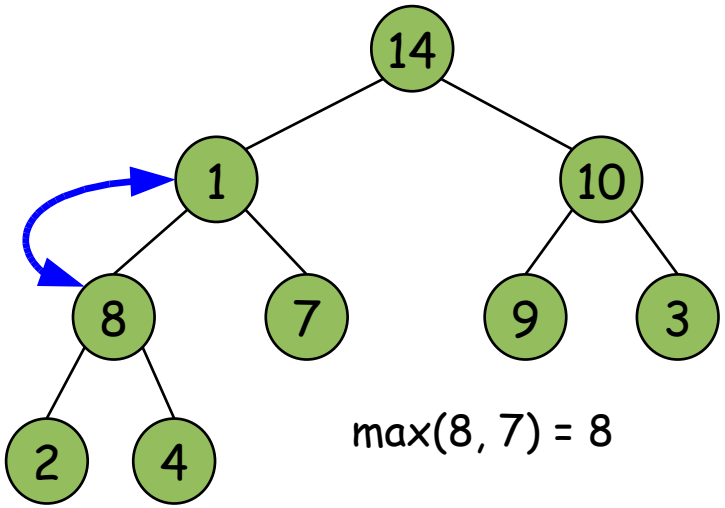


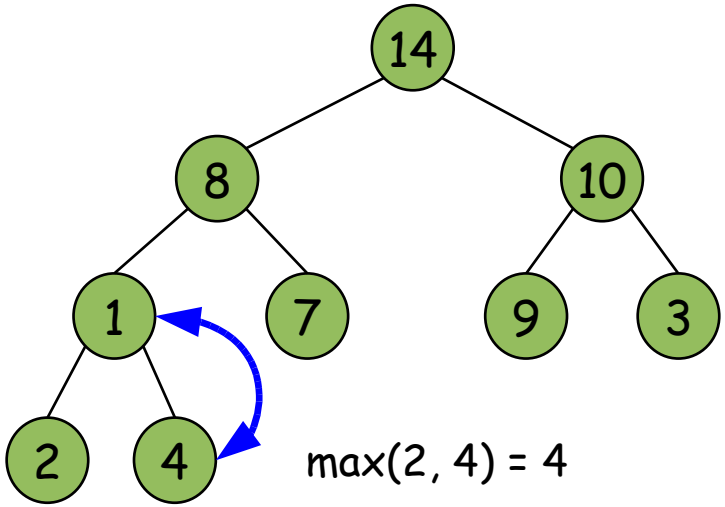


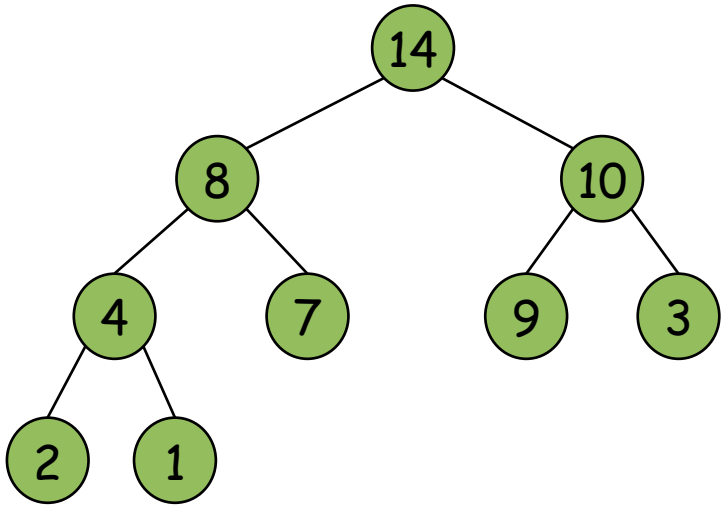
Facciamo scendere il nodo i nell'albero fino a trovare la sua posizione

Max-Heapify(A, i)









Max-Heapify(A, i)

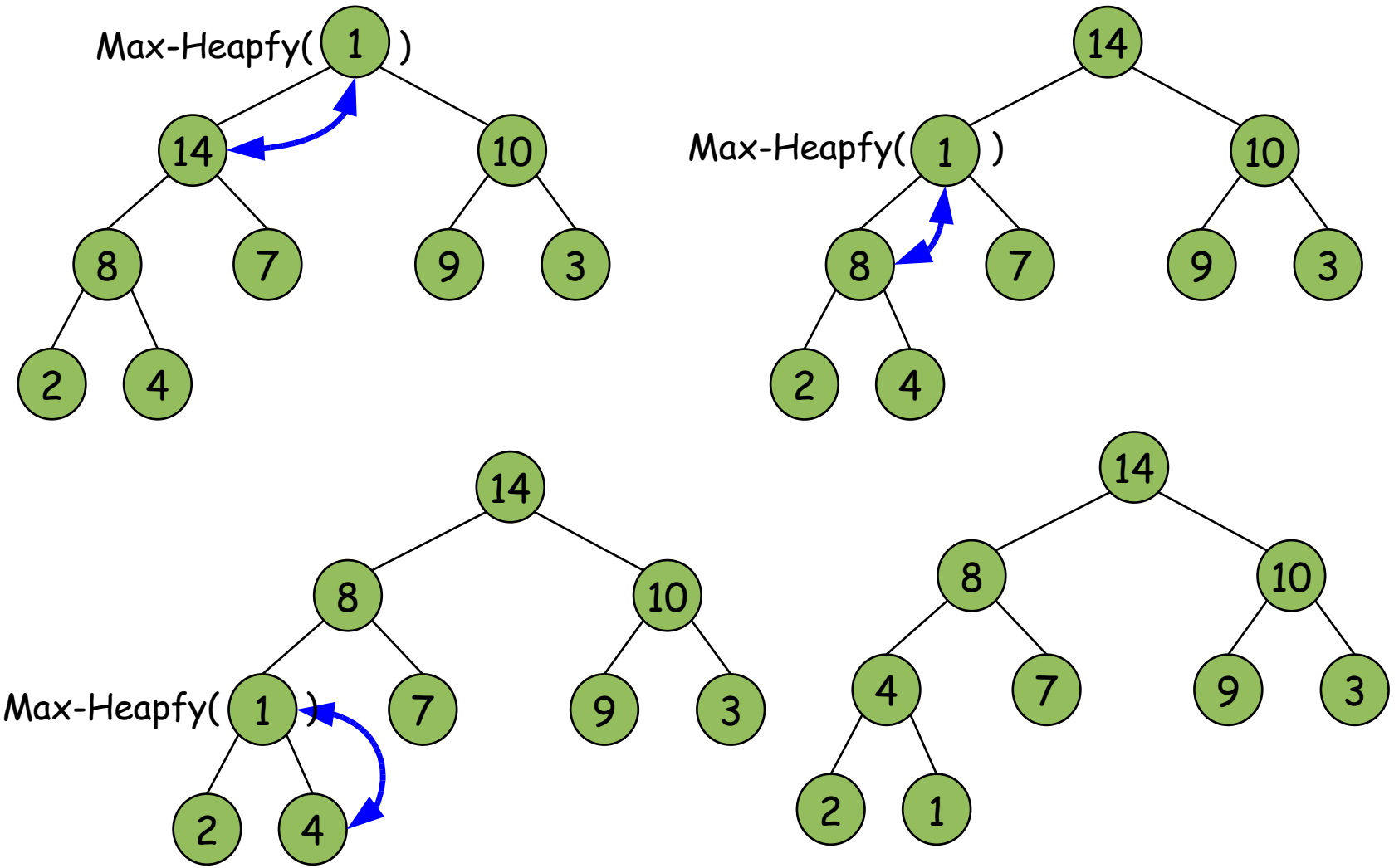
Max-Heapify(A, i)
  $l \leftarrow left[i]$
  $r \leftarrow right[i]$

  if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
    then $massimo \leftarrow l$
    else $massimo \leftarrow i$

  if $r \leq heap\text{-}size[A]$ and $A[r] > A[massimo]$
    then $massimo \leftarrow r$

  if $massimo \neq i$
    then scambia $A[i] \leftrightarrow A[massimo]$
      Max-Heapify($A, massimo$)

Max-Heapify(A, i)



Costo computazionale di Max-Heapify(A, i)

Il costo di **Max-Heapify** in un (sotto)albero di dimensione n è pari:

- al tempo costante $\Theta(1)$ necessario per stabilire il massimo fra gli elementi $A[i]$, $A[left(i)]$ e $A[right(i)]$
- più il tempo per eseguire Max-Heapify in un sottoalbero con radice uno dei figli di i

La dimensione di ciascun sottoalbero non supera $2n/3$

Il tempo di esecuzione di Max-Heapify può essere espresso dalla seguente ricorrenza

$$T(n) \leq T(2n/3) + \Theta(1)$$

Per il teorema master, la soluzione di questa ricorrenza è $O(\log_2 n)$

Esercizio

Risolvere la seguente ricorrenza usando il teorema dell'esperto

$$T_1(n) = \begin{cases} c + T_1(2n/3) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

In questo caso, il numero e la dimensione dei sottoproblemi sono pari ad $a = 1$ e $b = 3/2$ ($n/b = 2/3n$ implica $b = 3/2$). Inoltre, $\log_b a = \log_{3/2} 1 = 0$ e $f(n) = c = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_b a})$

Per il caso 2 del teorema dell'esperto abbiamo che

$$T_1(n) = \Theta(n^{\log_b a} \log_2 n) = \Theta(\log_2 n)$$

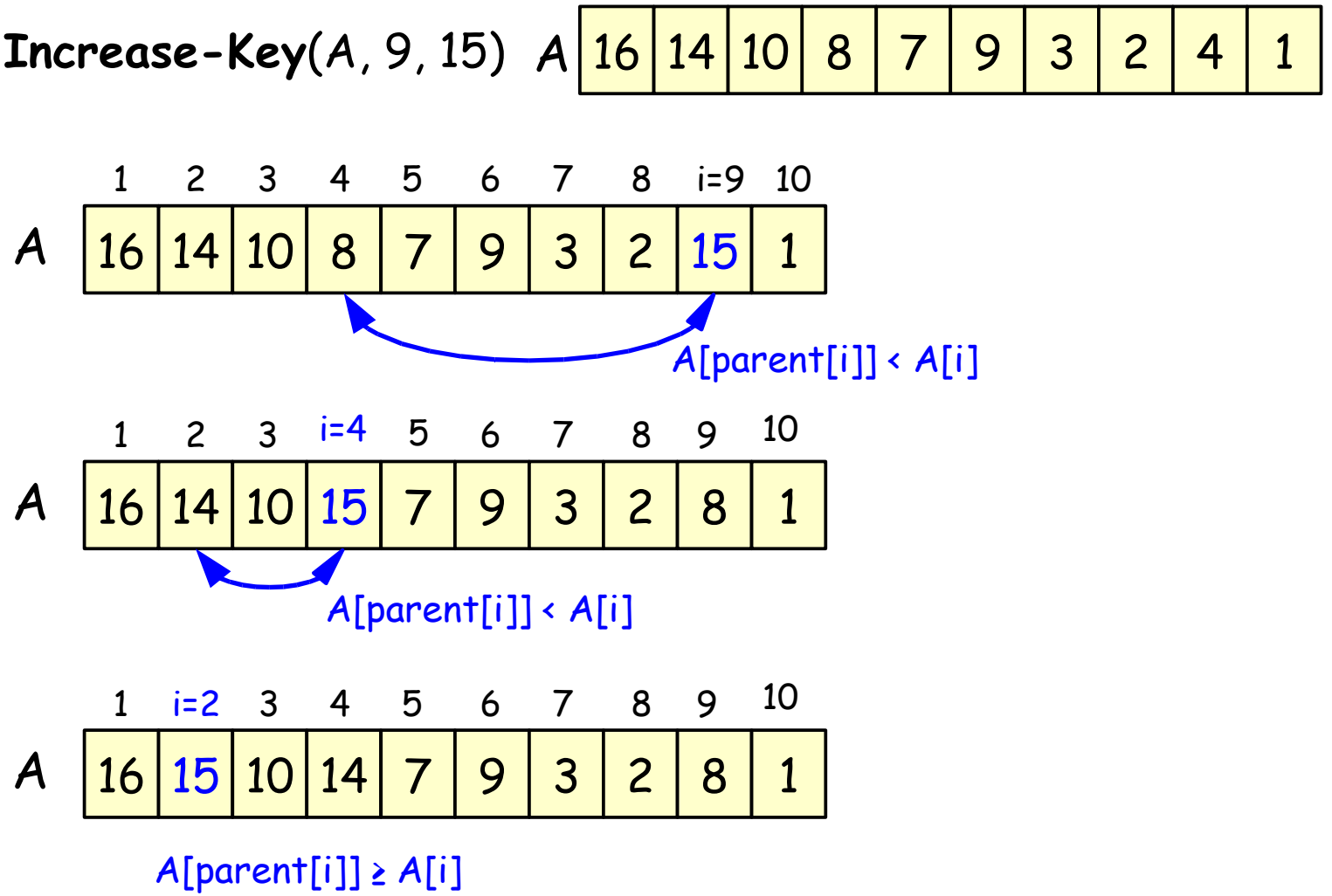
Estrazione del massimo

Heap-Extract-Max(A)
  if $\mathit{heap\text{-}size}[A] < 1$
    then error "underflow dell'heap"
  $\mathit{max} \leftarrow A[1]$
  $A[1] \leftarrow A[\mathit{heap\text{-}size}[A]]$
  $\mathit{heap\text{-}size}[A] \leftarrow \mathit{heap\text{-}size}[A] - 1$
  Max-Heapify(A, 1)
  return max

Il costo computazionale è dato chiaramente dal costo della procedura Max-Heapify, ossia $O(\log_2 n)$

Incremento di una chiave

Increase-Key$(A, 9, 15)$



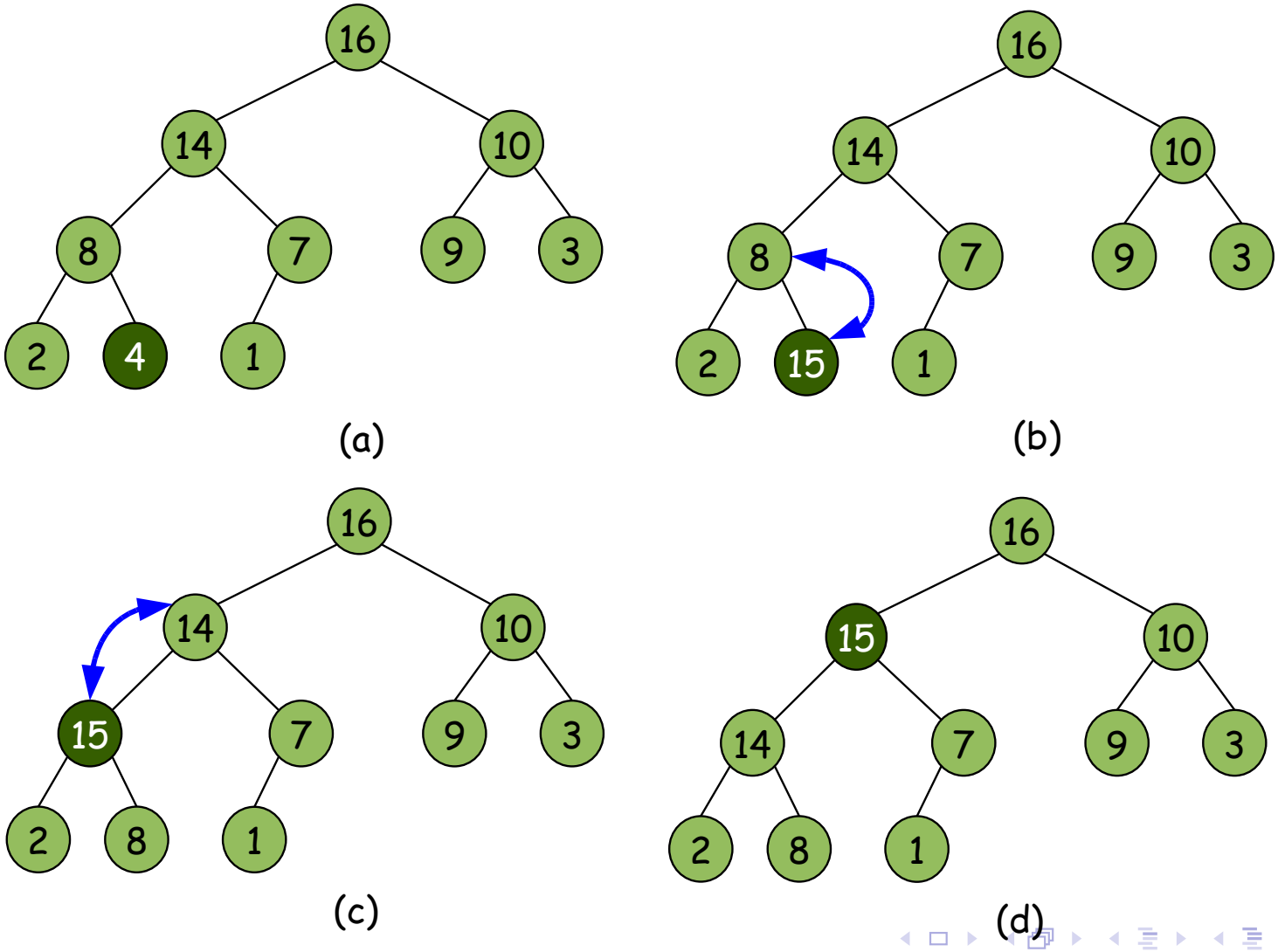
Heap-Increase-Key(A, i, key)

È molto simile alla **Max-Heap-Insert**

Heap-Increase-Key(A, i, key)
  if $key < A[i]$
    then error "la nuova chiave è più piccola di quella corrente"
  $A[i] \leftarrow key$
  while $i > 1$ or $A[parent[i]] < A[i]$
    do scambia $A[i] \leftrightarrow A[parent[i]]$
      $i \leftarrow parent[i]$

$$O(\log_2 n)$$

Incremento di una chiave



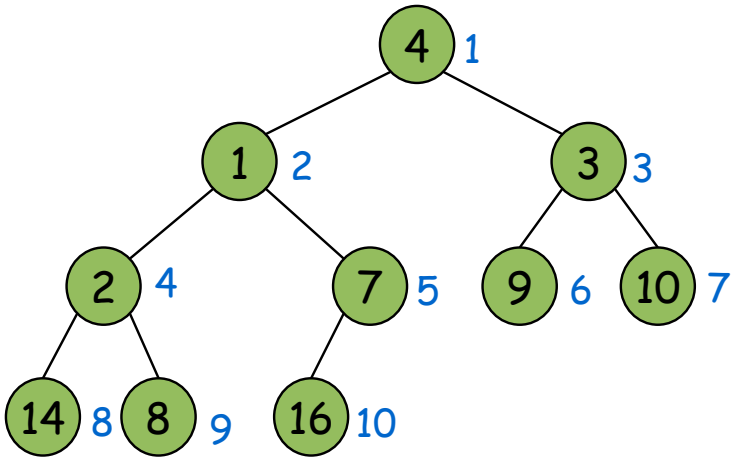
Costruzione di un heap

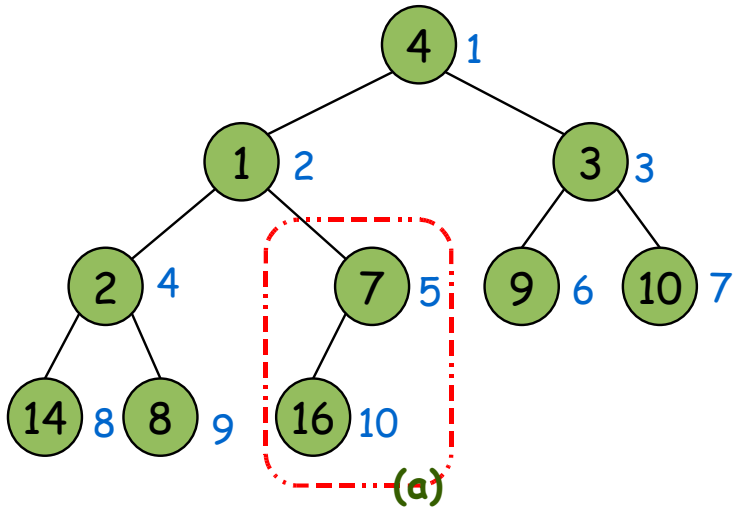
Possiamo usare la procedura **Max-Heapify** dal basso verso l'alto (in maniera bottom-up) per convertire un array $A[1 \ldots n]$ (non "heapizzato") in un heap

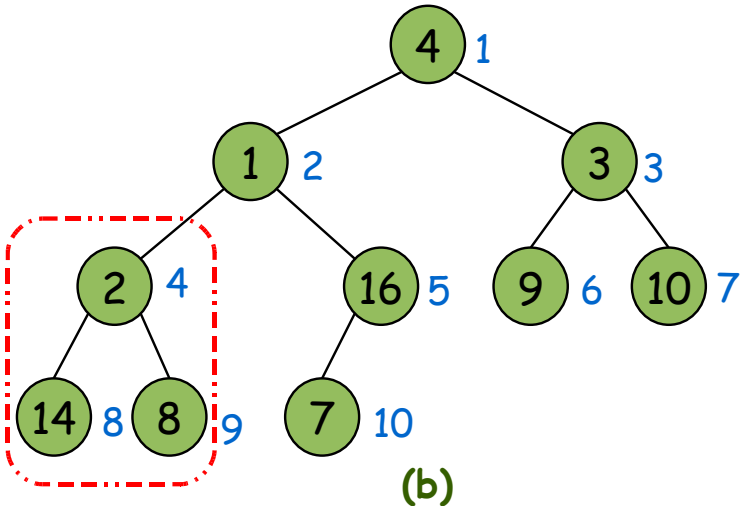
In un albero che rappresenta un heap di n elementi le foglie sono i nodi con indici $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$

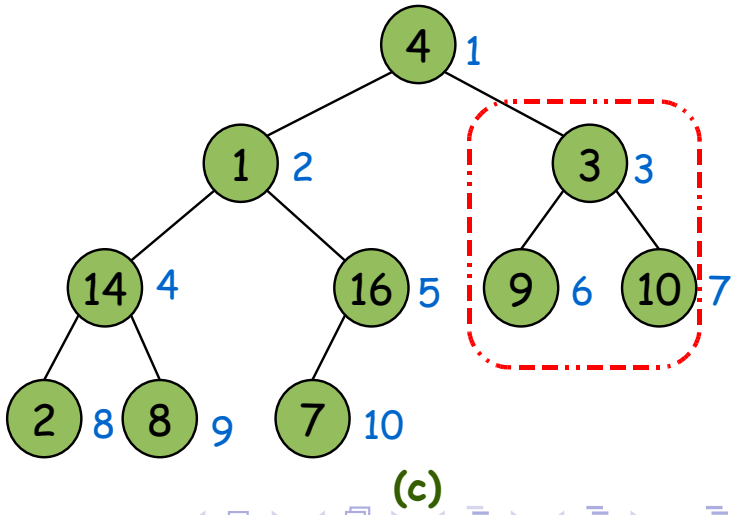
Ciascuno di essi è un heap di un elemento da cui cominciare

Costruzione di un heap

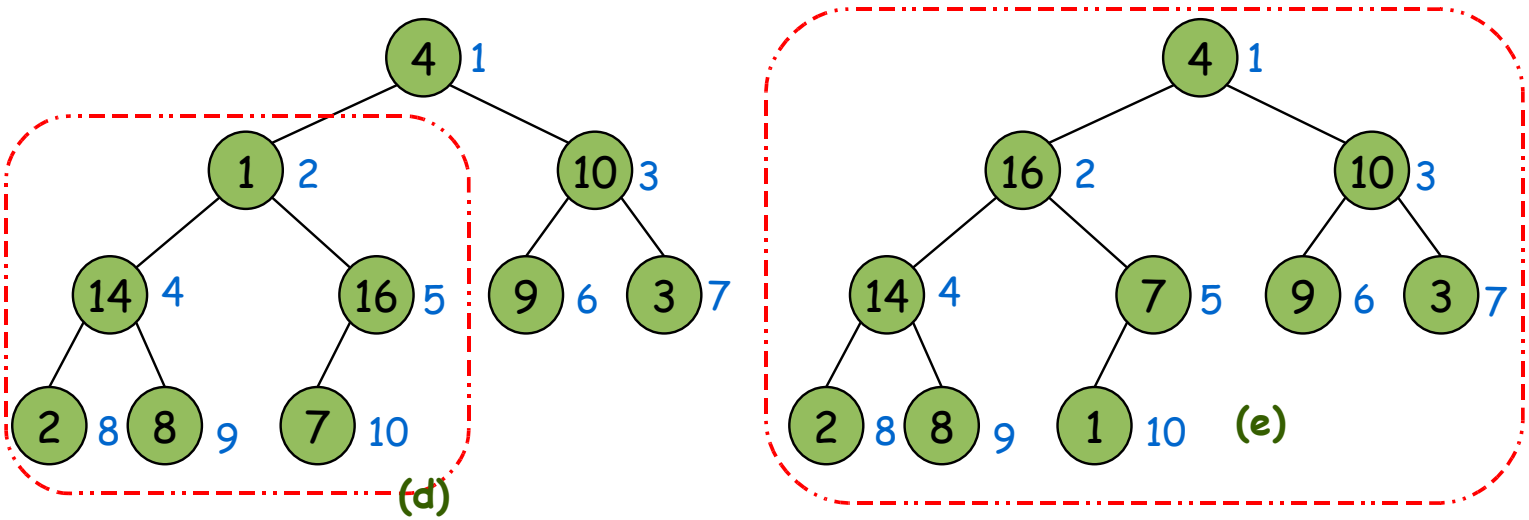


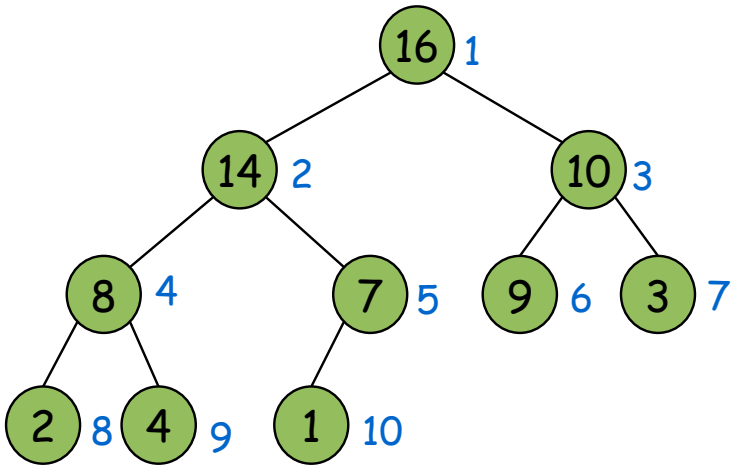






Costruzione di un heap





Build-Max-Heap(A)

Build-Max-Heap(A)
  $heap\text{-}size[A] \leftarrow length[A]$
  for $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
    do Max-Heapify(A, i)

Il tempo di esecuzione di ogni chiamata di **Max-Heapify** è (al più) $O(\log_2 n)$. Eseguiamo $\lfloor length[A]/2 \rfloor = O(n)$ chiamate di **Max-Heapify**. Quindi, il tempo di esecuzione è $O(n \log_2 n)$

Possiamo fare di meglio osservando che il tempo di esecuzione di **Max-Heapify** varia al variare dell'altezza del nodo nell'heap

Build-Max-Heap(A)

Un'analisi più rigorosa si basa sulle seguenti informazioni:

- uno heap di n elementi ha un'altezza $\lfloor \log_2 n \rfloor$
- per ogni $k = 0, \ldots, \lfloor \log_2 n \rfloor$ ci sono al più $\lceil n/2^{k+1} \rceil$ nodi di altezza k
- una chiamata di **Max-Heapify** su un nodo di altezza k è $O(k)$

Il costo $T(n)$ di **Build-Max-Heap** è limitato da

$$T(n) \leq \sum_{k=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{k+1}} \rceil O(k) = O(\sum_{k=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{k+1}} \rceil k)$$

poichè $O(f_1 + f_2) = O(f_1) + O(f_2)$

Build-Max-Heap(A)

Poniamo

$$f(n) = \sum_{k=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{k+1}} \rceil k \leq^{1} \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^k} k = n \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k}$$
$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k} = n \sum_{k=0}^{\infty} k(1/2)^k$$

poichè $\sum_{k=0}^{\infty} kx^k = x/(1-x)^2$, $\sum_{k=0}^{\infty} k(1/2)^k = 2$ e quindi

$$\sum_{k=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{k+1}} \rceil k \leq 2n = O(n)$$

[1] Poniamo $m = n/2^k$; allora $\lceil n/2^{k+1} \rceil = \lceil m/2 \rceil \leq \lceil m/2 \rceil + \lfloor m/2 \rfloor = m = n/2^k$

Build-Max-Heap(A)

Ricapitolando ...

se denotiamo con $T(n)$ il costo totale di **Build-Max-Heap**, allora

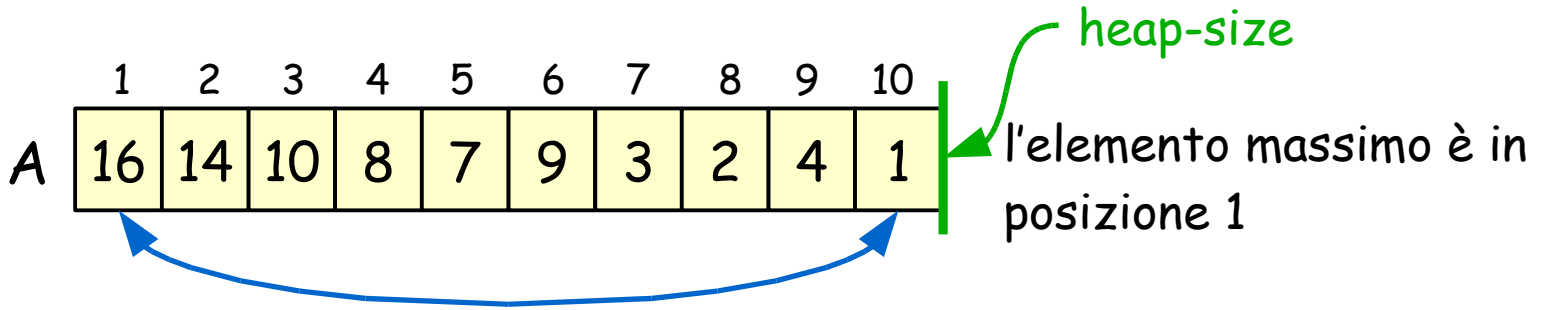
$$T(n) = O(f(n))$$

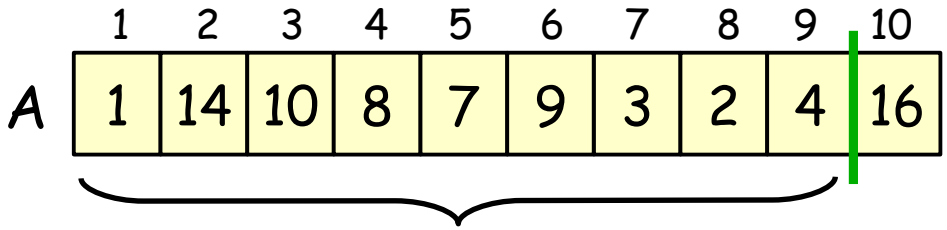
con $f(n) = \sum_{k=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{k+1}} \rceil k = O(n)$

Per la transitività della notazione O abbiamo che:

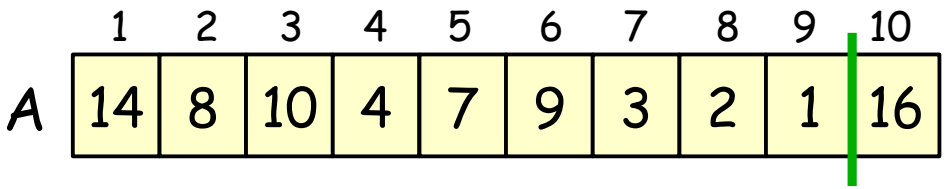
$$T(n) = O(n)$$

Ordinamento di un heap



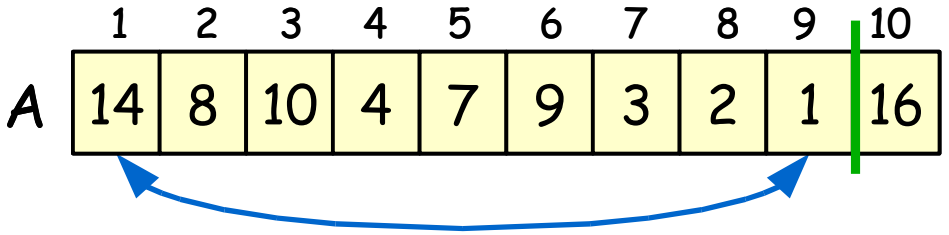


"heapizziamo" questa porzione di vettore

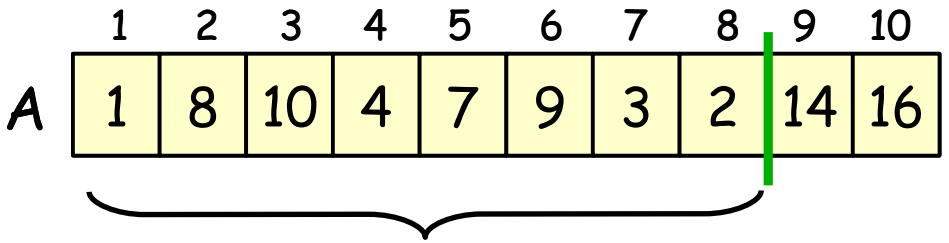


ed ... iteriamo

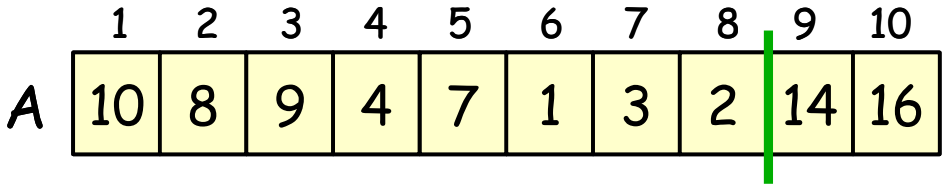
Ordinamento di un heap



il secondo elemento più grade è in posizione 1



"heapizziamo" questa porzione di vettore



iterando n-1 volte otteniamo un vettore ordinato

Heapsort(A)

Heapsort(A)
Build-Max-Heap(A)
  for $i \leftarrow$ *length*$[A]$ **downto** 2
    do scambia $A[1] \leftrightarrow A[i]$
      heap-size$[A] \leftrightarrow$ *heap-size*$[A] - 1$
      Max-Heapify$(A, 1)$

La chiamata di **Build-Max-Heap** impiega $O(n)$ e ciascuna delle $n - 1$ di **Max-Heapify** impiega $O(\log_2 n)$

La procedura **Heapsort** impiega un tempo $O(n \log_2 n)$